Applying Genetic Algorithms to Test JUH DBs Exceptions

Mohammad Alshraideh
The University of Jordan, King Abdullah
II School for Information Technology,
Department of Computer Science, Amman 11942 Jordan,

EzdeharJawabreh
Palestine Polytechnic University,
Applied Science Department

Heba M. AL Harahsheh

Basel A. Mahafzah
The University of Jordan, King Abdullah
II School for Information Technology,
Department of Computer Science, Amman 11942 Jordan,

***Abstract*— Database represents an essential part of software applications. Many organizations use database as a repository for large amount of current and historical information. With this context testing database applications is a key issue that deserves attention. SQL Exception handling mechanism can increase the reliability of the system and improve the robustness of the software. But the exception handling code that is used to respond to exceptional conditions tends to be the source of the systems failure. It is difficult to test the exception handling by traditional methods. This paper presents a new technique that combines mutation testing and global optimization based search algorithm to test exceptions code in Jordan University Hospital (JUH) database application. Thus, using mutation testing to speed the raising of exception and global optimization technique in order to automatically generate test cases, we used fitness function depends on range of data related to each query. We try to achieve the coverage of three types of PL/SQL exceptions, which are No_Data_Found (NDF), Too_Many_Rows (TMR) and Others exceptions. The results show that TMR exception is not always covered this due to existence of primary key in the query, also uncovered status appear in nested exceptions.**

Keywords—Database Application; Exception Handling;Mutation Testing; Genetic Algorithms; Select Statement.

I. INTRODUCTION

Dynamic test data generation methods use information from the execution of the program under test. A simple example of a dynamic method is random test data generation [21]. In this method, candidate test data is generated randomly by sampling values from the input domain. Each candidate test case is then executed and only those that cover a required program element are retained.

The problem with this approach is clear with complex programs or complex adequacy criteria, where an adequate test input may have to satisfy very specific requirements. In such a case, the number of adequate inputs may be very small compared to the total of inputs, so probability of selecting an adequate input by chance can be low. As an example, consider the problem of generating an input to execute the target branch A of the Flag program shown in Fig.1. The target branch A is executed when a = 1. The problem is to find input values, x and y such that a is set to 1. Note that f may be a complex and poorly understood function of x and y. Depending on the size of the domains of x and y and on the behaviour of f, it is possible that there is only a very small probability that a randomly generated input will set the variable a to 1 and thus execute the target branch A in Fig.1 In general, random test data generation is generally considered to be ineffective at covering all branches in realistic programs [22].

```
void Flag(int x, int y)
{
        a = f(x, y);
        if (a == 1)
                flag = true;  // target A
                        }
        ...
        if (flag) {
                // target B
                }
}
```

Fig.1. An example of a program (called Flag) with a "flag variable" problem.

Search-based software testing is a dynamic method of test data generation in which search methods or optimization techniques are used to generate tests and have been successfully applied in structural testing [[24] [25[23] [26][27][28][29]]. As with all search methods, search based software testing relies crucially on an evaluation or cost function to compare candidate test cases.

Database application is an important class of software that requires intensive testing. Usually database application is defined as a program that communicates with data stored in the database. Typically, this communication is done by using Structured Query Language (SQL). SQL includes Data Definition Language (DDL) that creates database schema or integrity constraints and Data Manipulation Language (DML) that retrieves or modifies records stored in the database, such

as SELECT, INSERT, DELETE, MERGE and UPDATE statements.

An important part of any database application that requires testing is the exceptions handling part. This part of code is responsible for recovering system from unusual events (exceptions) that may occur while communicating with the database. It also helps in designing more modular programs since it can be placed separately. Therefore, it is important to test this code effectively. The test period is the most important period in the software life and its cost is the biggest. In the study of Sinha and Harrold [19], they observed that 23.3% and 24.5% of the classes contained try and throw statements. If the efficiency of testing exception handling is improved, then it will improve the efficiency of the testing of the whole system.

Although the purpose of the exception handling code is to improve the robustness of the software, people noticed that the exception handling code contains more errors than the other parts of the software. For example, in a case-study by Toy [2], more than 50% of the operational failures of a telephone switching system were due to faults in exception handling and recovery algorithms. Another example, the Ariane 5 launch vehicle was lost due to an un-handling exception destroying $400 million of scientific payload [1].

Generally, exceptions' handling testing require methods that are different from the usual ones. For example, to test exceptions handling code in conventional program, a research by Tracey et al. [1], used a technique that is based on global optimization algorithm to automatically generate test cases for the purpose of testing the handling of runtime exceptions in safety critical systems. Other method by Zhang [3], presented a mutation technique to accelerate the raising of exceptions in order to save time spent for exception to occur.

To illustrate the problem, consider the problem of generating test data to execute the target no_data_found exception in Fig.2. The PL/SQL block in Fig. 2 selects information about the salary and commission of employee whose number is 102055. If no information is found it raises the No_Data_Found exception and the two variables assigned to zero. Note that No_Data_Found is a system defined exception.

```
Begin
        select salary, commission into v_salary,
v_commssion
        from employee where  employee_number = 102055;
Exception
                whenNo_Data_Found then
                v_salary = 0;
v_commission =0;
End;
```

Fig.2. An example of a program exception.

According to our knowledge, no research till now has focused on testing exceptions handling in database application. For this purpose, this research presents an approach that covers exceptions code in PL/SQL Oracle 10g database for JUH application. Our approach takes benefits from ideas used in conventional exceptions handling testing. Also, our research work rephrases the idea of mutation testing used by Zhang [3] and global optimization searching algorithm used by Tracey [1], and integrates them in testing exceptions that results from querying the database; i.e. from SELECT statements. In our research, three types of exceptions where studied: No_Data_Found (NDF), Too_Many_Rows (TMR) and others.

The rest of the paper is organized as follows: Section 2 discusses the background and related work. Then, in Section 3, the implementation of our system is described. Section 4 shows the conducted experiments. The obtained results are discussed and analyzed in Sections 5. Finally, Section 6 draws the conclusions and future research.

II. Background And Related Work

A common definition of exception is the union of "error," "exceptional case," "rare situation," and "unusual event" [17]. The entity that is raising an exception stops and waits for the completion of the exception processing. Exceptions are usually divided into two types: predefined and user-defined exceptions [18]. The predefined exceptions are declared implicitly and are also raised implicitly when the language rules are violated at run-time and in response to hardware errors. The user-defined exceptions are defined and detected at the application level; they can be raised explicitly in the application via the raise statements. Exception handling is the immediate response and consequent action taken to handle the exceptions. An exception handler is the code attached to an entity for one or several exceptions and is executed when any of these exceptions occur within the entity.

Test-data alone cannot test the raising of exceptions in response to hardware errors. In this paper, we focus on testing exception that violates run-time SQL rules (predefined). The input domain of most of programs, D, is likely to be very large, but the input domain which can causes an exception is likely to be small. It is very difficult to find the test-data that can raise an exception in large input domain. In this research, mutation is used only to give more paths to generate test cases.

A. Mutation Testing Overview

Mutation testing is a white-box fault-based testing technique originally proposed by DeMillo et al. [4]. The primary goal of mutation testing is to assist in developing adequate test suite, or it can be used to determine the effectiveness of a given test case by measuring its ability to detect faults. It operates by generating many versions (mutants) of the original program each has a fault that is injected by changing a syntactic operator (mutation operator) in the main program. Given the set of test cases to determine their effectiveness; they are initially executed against the original program to get the expected output [5], then they are executed against the mutants in the hope that they will give an output that is different from the original program's output. If this happen the mutant is said to be killed and the test case is an effective one, otherwise the mutant is a live and we need to generate more test cases to kill it. Some mutants will always give the same output as original program's output, which are called equivalent mutants.

One measure for this mutation process is mutation score which is the proportion of killed mutant over all mutants except equivalent ones. Mutation score supplies the tester with feedback about how the testing was completed and the adequacy of the test cases [5].

B. Mutation Testing for Database Applications

Many researches in testing database used mutation testing as a method to evaluate the effectiveness of the automatic test case generation techniques. In the literature Tsai et al. [6], introduced mutants by changing a set of mathematical operators in the code to determine the fault detection ability of the test cases. In testing database transitions with AGENDA, Chays and Deng injected faults in queries as a final step to demonstrate the adequacy of their proposed approach [7].

The most related works to this research are the researches which concerned with generating mutation operators specifically related for SQL queries. Chan et al. [8] proposed a mutation technique based on the conceptual data model. Their proposed technique employed the constraints that are in the Enhanced Entity Relation (EER) diagram to get SQL semantic based mutation operators. Tuya et al. [9] proposed a large set of SQL mutation operators designed for SELECT statement with the purpose of determining the adequacy of test suite and as a mean for injecting faults in order to compare different database testing techniques. The operators covered wide range of SQL particularities. Originally their mutants have four types: mutants for main SQL clauses (SELECT, JOIN, and subquery predicates), mutants for operators in the conditions or expressions, mutants related for NULL values and finally mutants for replacement of identifiers, as shown in Table 1. These mutants are further classified into types and subtypes, for more details see [9] [10]. Tuya and his colleagues tested the proposed mutants against a set of queries drawn from the NIST SQL conformance. In addition they tried to improve the feasibility of their mutants by running different experiments that aim to reduce the number of mutants (selective mutation) or to reduce the number of test cases (by reordering mutants). In this research, Tuya's mutants were studied and reused to test exceptions that raised from SELECT statements.

TABLE I. MUTATION OPERATORS USED IN THE EXPERIMENTS.

Category	Types
SC: SQL Clause Mutation Operators	SEL: SELECT clause.
	JOI: JOIN clause.
	SUB: Subquery predicates
	GRU: Group by clause.
	AGR: Aggregate functions
	UNI: Union, Union All.
	ORD: Order by clause.
OR: Operator Replacement Mutation Operators	ROR: Relational Operator Replacement
	LCR: Logical Connector Operator
	UOI: Unary Operator Insertion
	ABS: Absolute Value Insertion
	AOR: Arithmetic Operator Replacement
	BTW: Between predicate
	LKE : Like predicate
NL – NULL Mutation Operators	NLF: Null check predicates
	NLS: Null in select list
	NLI/NLO: Nulls in the input data
IR: Identifier Replacement Mutation Operators	IRC: Column replacement
	IRT: Constant replacement
	IRP: Parameter replacement
	IRD: Hidden column replacement

C. Genetic Algorithm

Genetic Algorithm (GA) was developed initially by Holland in the 1960s and 1970s [11]. It is a heuristics global optimization technique that attempts to find a good approximation to the optimal solution in a search problem. GA simulates the evolutionary process through the implementation of selection, recombination and mutation processes.

In literature GAs were used to automate the process of test data generation by searching the domains of the applications for suitable values that meet some testing criteria. Tracey et al. [1] used a GA to automatically generate test cases for testing exception handling code in safety critical systems. The technique is based on a global optimization technique; a GA that searches for the closest test data that will cause a run time violation to occur. Khor and Grogono [12] introduced genet, which is an Automated Test Data Generator (ATG) to generate test data for branch coverage. Their technique did not use program graphs because it was programming language independent and it used a GA to search for test data in the variables domains.

Masud et al. [13] introduced a strategy for mutation testing using GA, in which they instrument mutants, divide the program into small unit and then try to kill each mutant unit using genetic algorithm with special fitness function. Domínguez-Jiménez J. et al. [14] proposed a framework for mutant genetic generation for WS-BPEL (an XML language for web services). The approach defines a set of mutation operators for WS-BPEL. It automatically generates mutants using a GA that reduces the computation cost of executing by selecting a set of possible mutants. Bottaci [20] proposed fitness function which is defined in a way that a test case is able to kill a mutant if it satisfies the same three conditions used by Offutt in CBT [5], namely, the reachability, the necessary and the sufficiency conditions.

D. Genetic algorithm fitness function

The fitness functions in GA used for testing coverage three types of exceptions are as follows: Maximum and minimum values used as fitness to test coverage of NDF exception, if the value that generated by GA for column in the query or mutant under testing greater than the maximum value of this column in the database or less than the minimum value then NDF exception converge in this query. So, the fitness value is equal the min [distance between test case value and maximum column value, distance between test case value and minimum column value] +1.

Fitness for TMR is COUNT for the value that generated to column for query, if the value that generated appears two times or greater and that depending of structure of query then TMR exception coverage. So, the fitness is equal the distance between test case value and closed column value has count greater than 1. Table 2 shows an example of Testtable to illustrate how to calculate the fitness function to cover exception types. Three columns in Testtable (C1, C2 and C3) with number data types, these columns values have different values.

TABLE II. TESTTABLE VALUE FOR THREE COLUMNS C1, C2, AND C3.

C1	C2	C3
1	4	90
-20	-890	54
1	56	843
2601	786	-30
-5200	2000	1800
-20	199	-234
400	213	-90

```
Begin
…
        Select C1 into localvarible from Testtable
        Where
                C1 =Testcase;
…
Exception
        When no_data_found then
                //Target1 Executed
        When too_many_rows then
                //Target2 Executed
        When Others
                //Target3 Executed
End;
```

Fig.3. Example of exception types.

Fig. 3 illustrated a query with 3 exceptions. To cover Target1 in Fig. 3, this means no_data_found exception need to be raised, if testcase = 700 for example, then the fitness value = min((2601 -700), (700 – (-5200))) +1, = min(1901, 5900) +1= 1902, So the cost value is 1902 to execute Target1. If we need to execute Target2 in Fig. 3 with the same Testcase (700), then the fitness value will be calculated as follows: Fitness value = the difference between 700 and the closed column value which has frequency more than 1. Which is equal to abs(700 -1) =699. Table 3 shows test cases examples and NDF and TMR fitness values.

TABLE III. EXAMPLES ILLUSTRATED FITNESS FUNCTION FOR THE LISTED VALUES.

Test case value	NDF Fitness value	TMR Fitness value
700	1902	699
100	2501	99
1	2601	0
3000	0	2999
-2000	3001	2980

E. Jordan University Hospital Computer System

The core of Jordan University Hospital (JUH) information system is bought in 1994, and then the JUH IT team developed the Hospital Information System (HIS) using Oracle forms, and upgrades it to Oracle 10g. HIS developed to provide best medical services for patients and physicians. Delivering these services require hospitals to review the way they manage their business processes and supply more efficient features to physicians, patients, and hospitals officials as well as other decision makers. In order to provide such services, the health facility must focus on developing a solution to connect all its resources and makes it available to all who needs utilizing it using latest technology. This kind of solution will enhance the performance and optimize the efficiency and will reduce the cost of ownership.

IT department in JUH creates a solution suite that transforms the hospital to a community allowing the access to all resources and data as needed. HIS is a comprehensive solution developed specifically for health facilities in the region. It is flexible, comprehensive, multilingual, integrated and secured solution that supports clinical, financial, administration and higher management needs.

In general, a hospital management system can be sub-categorized into the following groups (Fig. 4):

- Medical Information System (Administrative and Clinical).
- Enterprise Resource Planning (ERP) (Material, Financial and Human Resources).
- Support System.

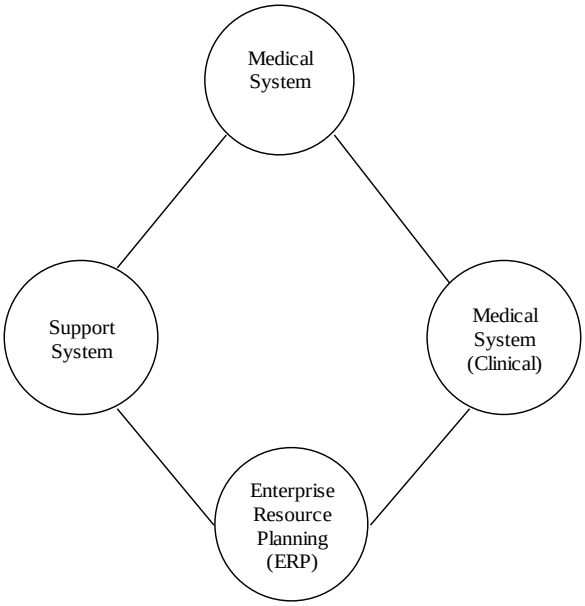

Fig.4. Hospital management system sub-categorizes.

Medical systems are developed to deliver all needed services to the hospital community (physicians, patients and administration). The systems manage all patients' data and information during their treatment episode in a professional and efficient manner. Medical systems strategically support a full range of hospital functions. It contains a repository of all patients' clinical, billing and demographic data, reducing paper work, manual effort and errors. Furthermore; it allows for better staff utilization allowing for more time to focus on planning and goals achievements. This enables the hospital to provide better quality and more efficient services, needed by patients and physicians. Medical systems are integrated with financial, administration, human resources, and material management systems. It contains vast collection of data including patient data, treatment data, hospital visit data, patient transactions data, hospital data, and statistical information.

HIS medical systems provide many key functions including:

- Medical administrative including:
  - Patient master index
  - Admission, discharge and transfer
  - Scheduling and appointments
  - Medical records
  - Medical reports
  - Medical statistics
  - Catering
  - Order entry and results communication
- Medical clinical including:
  - Out-patient clinics
  - Accidents and emergency
  - Operation theater
  - Maternity
  - Doctors desktop
  - Nurse station
  - Laboratory
  - Radiology
  - Pharmacy
- Patient accounting including:
  - Pricing and package deals
  - Patient billing
  - Insurance contract management
  - Claims management.

In order to test our system in this research we will select different procedures and functions, which will be described later in this paper.

III. Proposed System Overview

Fig. 5 depicts the basic structure of our system. Following is a brief description of the system's components:

- **SQL Mutation Tool:** which is a web service tool designed by Tuya et al. [9]. This tool accepts as inputs the SQL query and DB schema file. It will generate a set of mutants for the given query according to predefined SQL mutation operators as suggested by Tuya, as seen in Table 1. The DB schema and queries in our system were related to Human Resources (HR) sample database in Oracle 10g.

- **Mutant's Pool:** the required numbers of queries that satisfy a set of conditions are saved in the mutant pool waiting to be processed by the next stage.

Extracting Parameters: the SQL mutation tool supports the existence of parameters in the SQL query. For example, in the following query,there are two parameters *x* and *y* of type integer, one for location_id and the other for department _id. The pair of question marks symbol denotes the existence of parameter as it supported in the SQL Mutation Tool. The letter just before the last question mark indicates the type of parameter, where *I*stands for integers, *d* for decimal, *c* for characters and strings and *u* for date data type. This stage will extract parameters information and packed it in a record called TC structure, in the following query it will be (integr, integr).

SELECT manager_id from departments where location_id=?xi? ordepartment_id=?yi?

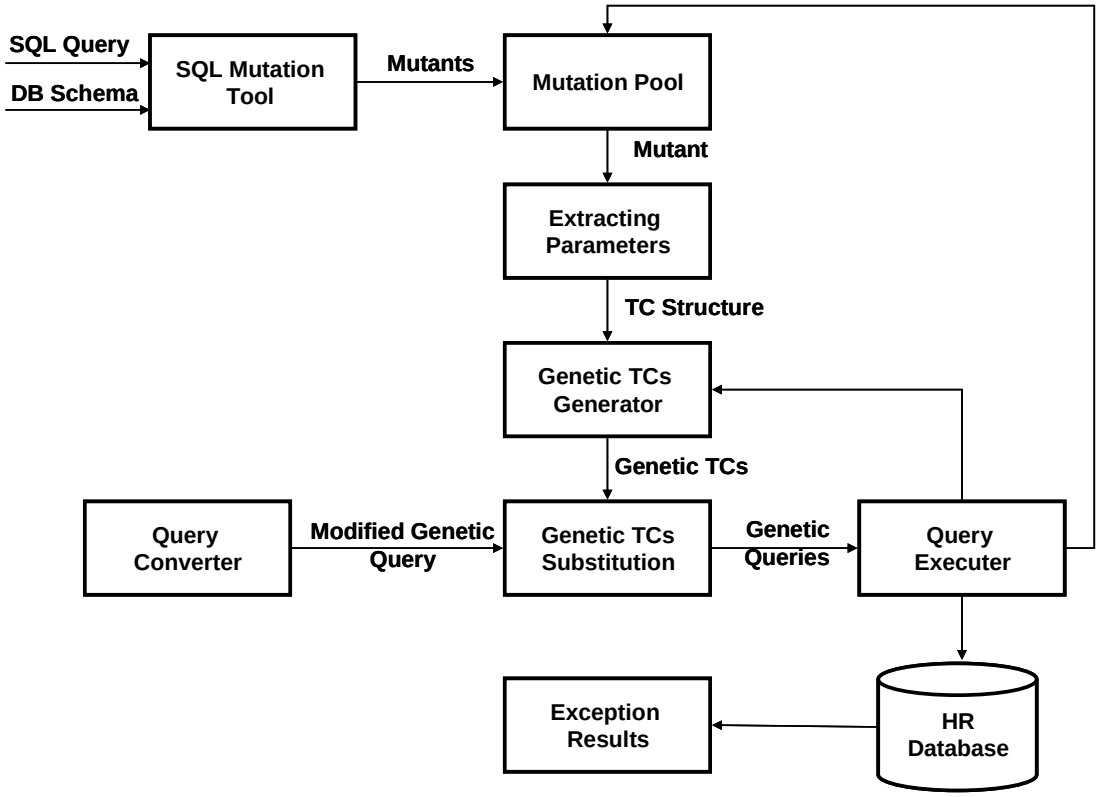

Fig.5. Basic structure of our system.

- **Genetic TCs Generator:** for each mutant TC structure, a GA will generate random population of the appropriate type within some specified range. The initial population consists of ten individuals (test cases). The population in the successive generations is resulted from crossover and mutation processes. The GA used here has no fitness function, so all the initial random ten individuals are used to participate as parents. To generate the offspring, a one point cross-over operation is performed between each pair of parents. The resulted offspring is then randomly mutated and used as the new population for the next generation. After performing many experiments to find the suitable TC, we found that running the GA for fifty generations will achieve the desired coverage of the intended exceptions. This TC called genetic test case.
- **Query Converter:** the mutants generated by SQL mutation tool are just SELECT statements and in order to raise any of NDF or TMR exceptions, a modifier is used in this system to convert SELECT statement into SELECT INTO statements. This is done by performing the necessary steps of defining the required number of variables according to the SELECT list using the schema file to extract variables types.
- **Genetic TCs Substitution:** for the mutant who is currently processed and modified, the parameters in the mutant will be substituted by the corresponding ones in each of the genetic TCs. The resulted query for a given mutant is called modified genetic query.
- **Query Executer:** the executer uses the modified genetic queries as input. Its main operation is to replace them one by one in PL/SQL block that contains exceptions handling code for the three exceptions: NDF, TMR and Others as shown in Fig. 6.

```
Declare
        Variable Definition List from preprocessor
Begin
        The modified genetic query
Exception
        whenno_data_found then
```

```
                …
:NDFFlag:=1;
        whentoo_many_rows then
         …
        :TMRFlag:=1;
        when others then
         …
        :OthersFlag:=1;
End;
```

Fig.6. Exception flag setting

The overall process of our system can be summarized in Fig. 7. The algorithm in Fig. 7 will iterate until the population has evolved to form a solution to the problem (i.e., solutions that achieved coverage of three intended exceptions), or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

```
For each SQL Query generate all possible mutations and save the required ones
in pool P.
For each mutant m in P and stop criteria not met do {
        Extract TC structure from m.
        Convert SQL SELECT in m to SELECT … INTO … command.
Simple Genetic Algorithm ( )
  {
        Initialize TC population;
        Evaluate population;
        While termination criterion not reached
        {
                Select solutions for next population;
                For each solution s in the selected population
                {
                        Substitute s in mutant m;
                        Execute mutant m and watch the result if any
        exception appear.
                        Exit if the flags of three exceptions are set.
                }
        Find next population by performing crossover and mutation;
        Evaluate population;
        }
  }
}
```

Fig.7. Algorithm used to find test case to coverage SQL Exceptions

IV. EXPERIMENTAL ENVIRONMENT

A. Test Objects

This section describes the test objects and the input domain sizes used. The following are source code for the test objects:

- ***OutPricing:*** Determines the pricing of treatments at outpatient clinics. Depending on his insurance the patient, this function calculates the amount of money the patient has to pay (depending on the type of insurance, the patient pays different ratios for his treatment) and the amount of money the insurance has to pay for the patient's treatment as shown in Fig. 8.
- ***InPricing:***This procedure calculates the invoice value of the patient inside the hospital based on the type of patient insurance and the type of medical procedure offered to patients (accommodation, scouting, doctors' fees, operations, laboratory, radiology, medicine, etc.). Also, this program calculates the percentage paid by the patient and the percentage paid by the insurance company, if any. Moreover, this function bills the patient with the amount of money he has to pay and bills the insurance company with the amount of the money it has to pay.
- ***JU-Med-fees-deduction:*** This package used for Jordan University staff, where there is an allocated account number for each staff in the system of JUH. It calculates bill value based on Jordan University insurance. Then deported the total amount of bill after deduct the hand-collect from the patients into tables to be used in Jordan University financial department later on.
- ***at-info-ibr:*** This function calculates the invoice value for private patients (in patients and out patients), then bills the patients with the amount of money he or she has to pay.
- ***Lab-interface:*** The main goal of this function is to transfer the results from medical machines (lab devices) to HIS system automatically (without user interaction). So, the function receives the message from medical devices then converts it to be entered to HIS system.
- ***salup_new_calc_all:*** This procedure calculates staff incentives as follows: it selects the category that owns the nursing, administrative, officer or a medical technician, by the department and qualifications. Then it determines the share of the incentives that the employee is entitled, as his career (Branch Chief, Chief, Division of, etc.). It discounts days leave without pay from the employee share incentives as shown in Fig. 9.

```
[1]  select prc_division,prc_category into p_div,p_cat from store_pricing_groups where
[2]  div_id=p_divn  and  grp_id=p_grp
[3]  …
[4]  select decode(p_insur_type, '1', prc_limit_out_e, '2', prc_limit_out_f)
[5]      into p_max_cov
[6]      from prc_limits
[7]      where prc_group    = p_group_id
[8]        and prc_division = p_div;
[9]  p_max_cov := nvl(p_max_cov, 99999);
[10]  Exception
[11]    when no_data_found then
[12] p_error_no := 1; raise exit_proc;
[13]    when others then
[14] p_error_no := 11; raise exit_proc;
[15] ….
[16] when exit_proc then
[17]    if p_error_no = 1 then
[18] raise_application_error( -20001, ‘Coverage limitsdo not exist.');
[19] elsifp_error_no = 2 then
[20] raise_application_error( -20003, ‘Ratepricingdoes not existforthis materials!!');
[21] elsifp_error_no = 3 then
[22] raise_application_error( -20004, 'Materialis not definedin the tableprice!!');
[23] elsifp_error_no = 5 then
[24] raise_application_error( -20006, ‘Pricing datais incompleteforthis patient!!');
[25] elsifp_error_no = 11 then
[26] raise_application_error( -20001, '11');
[27] elsifp_error_no = 21 then
[28] raise_application_error( -20003, '21');
[29] elsifp_error_no = 31 then
[30] raise_application_error( -20004,  '31');
[31]    end if;
[32]  when others then
[33] raise_application_error( -20005,   sqlerrm);
```

Fig.8. An example of a program (exception) **OutPricing**.

```
[1]  …
[2]  SELECT chng_date, old_emp_admin, old_emp_department, old_emp_job
[3]  into v_chng_date, v_old_emp_admin, v_old_emp_department, v_old_emp_job
[4]  FROM EMP_JOB_CHANGES e
[5]   WHERE emp_id = salup_rec.emp_id
[6]     and CHNG_DATE BETWEEN TO_DATE('02-'||TO_CHAR(P_MONTH-5, '00')||'-'||
[7]  TO_CHAR(P_YEAR, '0000'), 'DD-MM-YYYY')
```

```
[8]          and salup_rec.emp_join_date<chng_date
[9]      Exception
[10]     When no_data_found then
[11]         Begin
[12]           select days into v_leave_days
[13]           from salup_new_leaves_200806
[14]         where emp_id = salup_rec.emp_id
[15]                and month = p_month
[16]                and year = p_year;
[17]                and salup_rec.emp_status<> 2;
[18]          exception
[19]              when no_data_found then
[20]     v_leave_days:= 0;
[21]          end;
[22]     When two_many_rows then
[23]     …
[24]     End;
```

Fig.9. Nested exceptions example (fragment of JU-med-fees-deduction program).

B. Hardware and Software Environments

In this section, the specifications of the experimental environment utilized by this work are presented. These specifications include both the hardware and software modules used in implementing the simulator. More specifically, the hardware specifications that are used in the experiments include a Dual-Core Intel Processor (CPU 2.66 GHz), 2 MB L2 Cache per CPU, and 1 GB RAM. Moreover, the software specifications that are used in the experiments include windows XP.

Moreover, in order to assess the reliability of the cost functions introduced in the previous section, an empirical investigation was done. A number of test procedures and functions were assembled from JUH and an attempt was made to generate inputs to achieve branch coverage. These programs are described in Table 4. The size of each program is given as Lines of Code (LOC), number of select statements, and number of exceptions (three types of exceptions), and the last column represents the total number of mutations for each program generated by the tool in [10]. The range of number of mutations for each Query is from 24 to 79 mutations.

The search was directed to generate data for one exception at a time. The order in which the exceptions of the program were targeted was arbitrary, except that no nested exception was targeted before the containing exception as shown in Fig. 9. This is not; in general, a good strategy since it will become stuck at an infeasible exception.

TABLE IV. THE FUNCTIONS USED FOR EMPIRICAL INVESTIGATION.

Program Name	Lines of Code	Number of Select Statements	Number of Exceptions	Total number of mutations
OutPricing	295	14	6	12138
InPricing	362	22	33	11222
JU-Med-fees-deduction	307	12	18	7368
Pat-info-ibr	259	9	9	11655
Lab-interface	1389	45	69	38892
Salup_new_calc_all	707	17	24	25453

A steady-state style genetic algorithm, similar to Genitor [24], was used in this work. The cost function values computed for each candidate input were used to rank candidates within the population in which no duplicate genotypes are allowed.

A probabilistic selection function selected parent candidates from the population with a probability based on their rank, where the highest ranking having the highest probability. More specifically, for a population of size n, the probability of selection (P_s) is shown in Equation 1.

$$P_s = \frac{2(n - rank + 1)}{n(n-1)} \qquad (1)$$

In this work, a fixed population size of 100 was used. This parameter was not "tuned" to suit any particular program under test. In a steady state update style of genetic algorithms (as used in this work); new individuals that are sufficiently fit are inserted in the population as soon as they are created.

The criterion to stop the search was set up to terminate the search after 50 executions of the program under test, when only if full coverage was not achieved. Individuals were recombined using binary and real-valued (one-point and uniform) recombination, and mutated using real-valued mutation. Real-valued mutation was performed using "Gaussian distribution" and "number creep". These queries (Select statements in program described in Table 4) contain 109 different SELECT statements related to JUH database with 159 exceptions as a whole. The list of the queries includes conditions of types: WHERE, HAVING, and ON. In the case of WHERE condition, different SQL clauses were implemented as: [not] BETWEEN, [not] IN, [not] LIKE, IS [not] NULL, Logical connector AND and OR and the using of expressions with relational operators (=, >, <, <=, >=, <>).

Each query along with the schema file was executed. The generated mutants for each one are saved, and then they are processed; each one in separate; by passing through the stages explained in our system in Section 3. In our experiments we executed all the resulted genetic queries for each mutant in order to assess the area of exceptions coverage as described in the algorithm in Section 3.

Table 5 shows all possible mutants for the first query 1 (line 1) in Fig. 8, where id: the identification numbers of each mutant. The same ids that are generated from sql mutation tool [10] are used. Mutant subtype: refers to type of mutants when it is applied to particular sql clause (for more information see table 1).

TABLE V. ALL MUTATIONS USED FOR (SELECT PRC_DIVISION, PRC_CATEGORY INTO P_DIV, P_CAT FROM STORE_PRICING_GROUPS WHERE DIV_ID=P_DIVN AND GRP_ID=P_GRP) QUERY.

ID	Cat	Type	Subtype	Mutated SQL
1	SC	SEL	SLCT	SELECT DISTINCT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
2	NL	NLS	NLSS	SELECT COALESCE(prc_division ,'9999') AS prc_division , prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
3	NL	NLS	NLSS	SELECT prc_division , COALESCE(prc_category ,'9999') AS prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
4	IR	IRC	IRCCS	SELECT STORE_PRICING_GROUPS.PRC_CATEGORY ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
5	IR	IRC	IRCCS	SELECT STORE_PRICING_GROUPS.DIV_ID ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
6	IR	IRC	IRCCS	SELECT STORE_PRICING_GROUPS.GRP_ID ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
7	IR	IRC	IRCPS	SELECT ?xc? , prc_category FROM store_pricing_groups WHERE div_id= ?xc? AND grp_id= ?xc?
8	IR	IRD	IRDDS	SELECT STORE_PRICING_GROUPS.INV_ITEM ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
9	IR	IRC	IRCCS	SELECT prc_division , STORE_PRICING_GROUPS.PRC_DIVISION FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
10	IR	IRC	IRCCS	SELECT prc_division , STORE_PRICING_GROUPS.DIV_ID FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
11	IR	IRC	IRCCS	SELECT prc_division , STORE_PRICING_GROUPS.GRP_ID FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
12	IR	IRC	IRCPS	SELECT prc_division , ?xc? FROM store_pricing_groups WHERE div_id= ?xc? AND grp_id= ?xc?
13	IR	IRD	IRDDS	SELECT prc_division , STORE_PRICING_GROUPS.INV_ITEM FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id= ?xc?
14	NL	NLI	NLIW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE (STORE_PRICING_GROUPS.DIV_ID IS NULL OR div_id = ?xc?) AND grp_id= ?xc?
15	NL	NLO	NLIW1	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE (STORE_PRICING_GROUPS.DIV_ID IS NULL OR NOT div_id = ?xc?) AND grp_id= ?xc?
16	NL	NLO	NLIW2	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE (STORE_PRICING_GROUPS.DIV_ID IS NULL) AND grp_id = ?xc?
17	NL	NLO	NLIW3	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE (STORE_PRICING_GROUPS.DIV_ID IS NOT NULL) AND grp_id = ?xc?
18	NL	NLI	NLIW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND (STORE_PRICING_GROUPS.GRP_ID IS NULL OR grp_id= ?xc?)
19	NL	NLO	NLIW1	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND (STORE_PRICING_GROUPS.GRP_ID IS NULL OR NOT grp_id= ?xc?)
20	NL	NLO	NLIW2	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND (STORE_PRICING_GROUPS.GRP_ID IS NULL)
21	NL	NLO	NLIW3	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND (STORE_PRICING_GROUPS.GRP_ID IS NOT NULL)
22	IR	IRC	IRCCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE STORE_PRICING_GROUPS.PRC_DIVISION = ?xc? AND grp_id= ?xc?
23	IR	IRC	IRCCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE STORE_PRICING_GROUPS.PRC_CATEGORY = ?xc? AND grp_id= ?xc?
24	IR	IRC	IRCCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE STORE_PRICING_GROUPS.GRP_ID = ?xc? AND grp_id= ?xc?

25	IR	IRD	IRDDW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE STORE_PRICING_GROUPS.INV_ITEM = ?xc? AND grp_id= ?xc?
26	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id<> ?xc? AND grp_id= ?xc?
27	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id> ?xc? AND grp_id= ?xc?
28	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id< ?xc? AND grp_id= ?xc?
29	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id>= ?xc? AND grp_id= ?xc?
30	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id<= ?xc? AND grp_id= ?xc?
31	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE (1=1) AND grp_id = ?xc?
32	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE (1=0) AND grp_id = ?xc?
33	IR	IRP	IRPCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = STORE_PRICING_GROUPS.PRC_DIVISION AND grp_id = ?xc?
34	IR	IRP	IRPCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = STORE_PRICING_GROUPS.PRC_CATEGORY AND grp_id = ?xc?
35	IR	IRP	IRPCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = STORE_PRICING_GROUPS.GRP_ID AND grp_id = ?xc?
36	OR	LCR	LCRW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? OR grp_id= ?xc?
37	OR	LCR	LCRW	SELECT prc_division , prc_category FROM store_pricing_groups WHERE (1=1)
38	OR	LCR	LCRW	SELECT prc_division , prc_category FROM store_pricing_groups WHERE (1=0)
39	OR	LCR	LCRW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc?
40	OR	LCR	LCRW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE grp_id = ?xc?
41	IR	IRC	IRCCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND STORE_PRICING_GROUPS.PRC_DIVISION = ?xc?
42	IR	IRC	IRCCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND STORE_PRICING_GROUPS.PRC_CATEGORY = ?xc?
43	IR	IRC	IRCCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND STORE_PRICING_GROUPS.DIV_ID = ?xc?
44	IR	IRD	IRDDW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND STORE_PRICING_GROUPS.INV_ITEM = ?xc?
45	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id<> ?xc?
46	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id> ?xc?
47	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id< ?xc?
48	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id>= ?xc?
49	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id<= ?xc?
50	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND (1=1)
51	OR	ROR	RORW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND (1=0)
52	IR	IRP	IRPCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id = STORE_PRICING_GROUPS.PRC_DIVISION
53	IR	IRP	IRPCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id = STORE_PRICING_GROUPS.PRC_CATEGORY
54	IR	IRP	IRPCW	SELECT prc_division ,prc_category FROM store_pricing_groups WHERE div_id = ?xc? AND grp_id = STORE_PRICING_GROUPS.DIV_ID

V. RESULTS AND EVALUATION

GA search strategy was investigated empirically by generating test data for the functions and procedures shown in Table 4. What is not clear, however, is how long such a search would take. In addition, the empirical investigation will provide information about the GA search.

The results in Table 6 show the average number of executions required to find exception coverage test data. Table 6 shows the total number of exception ineach program, the number of exceptions successfully converge by GA, the total mutations used to coverage these exceptions, and the number of executions required to find test data. From this table, we notice that not all exceptions are covered by GA, (**OutPricing, JU-med-fees-deduction, and Salup_new_calc_all)**,the type of exceptions that are not cover is like line 17 in Fig. 8, which represent branch condition in the exception, and line 19 in Fig. 9 which represents nested exceptions.

Table 7 shows the types of exceptions that are not covered by GA from Table 6. From Table 7 we notice that most type of mutations that are not covered is TMR mutation type. The reason that, the uncovered of TMR (the invisibly in TMR) is due to: the existence of primary keys in the query, or columns with unique values where it is impossible to get TMR in the result.

Table 8 shows the number of program executions to find test data for exception after excluding the exceptions that are not covered in Table 6. The number of program executions ranged from 10475 used 213 mutations in OutPricing program down to 1493 used only 49 mutations in InPricingprogram

TABLE VI. EXCEPTIONS COVERAGE RESULTS.

Program Name	Total Number of Exceptions	Number of Exceptions Successfully Coverage	Total Number of Mutations Used to Coverage Exceptions	Total Number of GA Generations
OutPricing	6	5	290	14325
InPricing	33	33	49	1493
JU-Med-fees-deduction	18	14	368	18134
Pat-info-ibr	9	9	123	3492
Lab-interface	69	69	181	7946
Salup_new_calc_all	24	21	545	16914

TABLE VII. TYPES OF NOT COVERAGE EXCEPTIONS.

Program Name	Total Number of Exceptions	Number of Exceptions Successfully Coverage	Types of Not Coverage Exceptions		
			NDF	TMR	Others
OutPricing	6	5	-	1	-
InPricing	33	33	-	-	-
JU-med-fees-deduction	18	14	1	3	-
Pat-info-ibr	9	9	-	-	-
Lab-interface	69	69	-	-	-
Salup_new_calc_all	24	21	-	3	-

TABLE VIII. EXCEPTIONS COVERAGE AFTER EXCLUDING NOT COVERAGE EXCEPTION IN TABLE VI.

Program Name	Number of Exceptions	Total number of mutations used to coverage Exceptions	Total number of GA generations
OutPricing	5	213	10475
InPricing	33	49	1493
JU-med-fees-deduction	14	164	7934
Pat-info-ibr	9	123	3492
Lab-interface	69	181	7946
Salup_new_calc_all	21	159	7614

Although the discussion and conclusions of our results were related to JUH database, it is applicable to any other database, since the presence of database constraints such as primary keys and the effects of mutation operators on the conditions will be the same regardless the contents of the database.

VI. Conclusions And Future Works

In this paper, we have designed a system to generate automatically TCs that cover three known exceptions (which are **No_Data_Found (NDF), Too_Many_Rows (TMR)** and **Others** exceptions) in PL/SQL Oracle database. This system combines the mutation testing in order to speed the raising of exceptions, with a genetic algorithm that will automatically generate TCs. Experiments have been done to evaluate the system on JUH database application. The obtained results were subject to analytical studies. The studies illustrate that not all exceptions are covered.

It is concluded that this system achieved the desired coverage of the intended exceptions. The TMR was the most difficult one to cover since it has a lot of reasons that make it an invisible path, such as: the existence of primary keys or unique values, the nature of the query and existence of different categories of mutants.

It is still believed that the interpretation of the obtained results needs more improvement in complicated exceptions or branch inside exceptions. The future works are based on extending this research work to solve these problems. However, ongoing researches have been established to improve the system in different areas, such as including other types of Oracle exceptions to cover.

References

[1] N Tracey., J. Clark, K. Mander , and J. McDermid,"Automated Test-Data Generation for Exception Conditions", Software-Practice and Experience, 30(1), 2000, pp.61-79.

[2] W. N.Toy, "Fault-tolerant design of local ESS processors, The Theory and Practice of Reliable System Design", D.P. Siewiorek and R.S. Swarz, eds., pp. 461–496, Bedford, Mass.: Digital Press, 1981.

[3] S. J Jiang.,Y. P Zhang, D. S Yan, and Y. P. Jiang,." An Approach to Automatic Testing Exception Handling", ACM SIGPLAN Notices, Vol. 40(8), pp. 34- 39, 2005.

[4] A. DeMillo Richard, D. S. Guindi, K. N. King McCracken , and A. Offutt Jefferson , "An Extended Overview of the Mothra Software Testing Environment", In Proceedings of the second Workshop on Software Testing, Verification, and Analysis, Los Alamitos, pp.142-151, 1988.

[5] A. DeMillo Richard and A. Jefferson Offutt, "Constraint-Based Automatic Test Data Generation", IEEE Transactions on Software Engineering, Vol. 17 (9), pp. 900-910, 1991.

[6] W.T. Tsai, D. Volovik, T. F. Keefe ," Automated test case generation for programs specified by relational algebra queries", IEEE Transactions on Software Engineering, 16(3), pp. 316-324, 1991.

[7] Chays David and Y. Deng , "Testing Database Transactions with AGENDA", In Proceedings of the 27th International Conference on Software Engineering (ICSE), IEEE Computer Society, pp.70- 87,2005.

[8] W. K Chan., S. C Cheung., and T. H Tse, "Fault-Based Testing of Database Application Programs with Conceptual Data Model", In Proceedings of the 5th International Conference on Quality Software (QSIC 2005), IEEE Computer Society Press, Los Alamitos, California, pp. 187-196, 2005.

[9] Tuya Javier, M. J. Suarez-Cabal and de la Riva Claudio, " Mutating database queries, Information and Software Technology" , vol. 49 (4), pp. 398 -417,2007.

[10] Tuya Javier, M. J. Suarez-Cabal, and de la Riva Claudio , "Sqlmutation: A Tool To Generate Mutants Of SQL Database Queries", In Proceedings of Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006), IEEE Computer Society Washington, DC, USA, p. 1, 2006.

[11] J. H. Holland ,"Adaptation in Natural and Artificial Systems", University of Michigan Press.1975.

[12] Khor S. and Grogono P. (2004). Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically, In the Proceedings of the 19th International Conference onAutomated Software Engineering, pp. 346-349.

[13] M. Masud, , A. M NayakZaman,., and N. Bansal, "Strategy for mutation testing using genetic algorithms", IEEE Electrical and Computer Engineering, vol. 13(2), pp. 1049 – 1052,2005.

[14] J. Dom´ınguez-Jim´enez ,A. Estero-Botaro, and I. Medina-Bulo,"A Framework for Mutant Genetic Generation for WS-BPEL", Springer-Verlag Berlin Heidelberg, Vol. 5404, pp. 229–240, 2009.

[15] Oracle. Oracle® Database Sample Schemas, http://download.oracle.com, [Online October, 2012].

[16] HYPERLINK "http://in2test.lsi.uniovi.es/sqlmutation"

i. http://in2test.lsi.uniovi.es/sqlmutation, [Online June, 2013].

[17] J. Lang and D. B. Stewart,"A study of the applicability of existing exception-handling techniques to component-based real-time software technology", ACM Transactions on Programming Languages and Systems, 20(2), pp. 274-301,1998.

[18] J. B. Goodenough,"Exception handling: issues and a proposed notation", Communications of the ACM, v. 18 (12), pp. 683-696, 1975.

[19] S. Sinha andM. J. Harrold, "Analysis and testing of programs with exception-handling constructs", IEEE Transactions on Software Engineering, 26(9), pp. 849-871,2000.

[20] L. Bottaci , "A genetic algorithm fitness function for mutation testing", In Proceedings of SEMINAL: Software Engineering using Metaheuristic Innovative Algorithms, Workshop 8, ICSE 2001, 23rd International Conference on Software Engineering, Toronto, Canada, pp. 3-7, 2001.

[21] J. E. Duran ,S. C. Ntafos, "An evaluation of random testing". IEEE Transactions on Software Engineering 10(4):438 – 444,1984.

[22] P. D. Coward, " Symbolic execution and testing". Information and Software Technique 33(1):53-64, 1991.

[23] R. Feldt, R. Torkar ,T. Gorschek, W. Afzal, " Searching for cognitively diverse tests: Towards universal test diversity metrics". In: IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW '08, Computer Society Washington, DC, USA, pp 178-186, 2008.

[24] M. Alshraideh M, and L. Bottaci , " Automatic software test data generation for string data using heuristic search with domain specific search operators". Software Testing, Verification and Reliability 16(3):175-203, 2006.

[25] M. Alshraideh , B. A. Mahafzah and S. Al-Sharaeh, "A Multiple-Population Genetic Algorithm for Branch Coverage Test Data Generation", Software Quality Control, Vol. 19, No. 3, pp. 489-513, 2011.

[26] M. Alshraideh,L.BottaciandB A.Mahafzah, "Using Program Data-State Scarcity to Guide Automatic Test Data Generation", Software Quality Control, Vol. 18, No. 1, pp. 109-144, 2011.

[27] M. Alshraideh ,B. A. Mahafzah., H. S. EyalSalman ,I. Salah , "Using Genetic Algorithm as Test Data Generator for Stored PL/SQL Program Units", Journal of Software Engineering and Applications, Vol. 6, No. 2, pp. 65-73, 2013.

[28] P. McMinn ,"Search-based Software Test Data Generation: a Survey: Research Articles. Software Testing", Verification & Reliability, Volume 14, Number 2, Pages 105-156, 2004.

[29] Korel B. (1990), Automated Software Test Data Generation. IEEE Transactions on Software Engineering, Vol. 16(8), pp. 870-879.